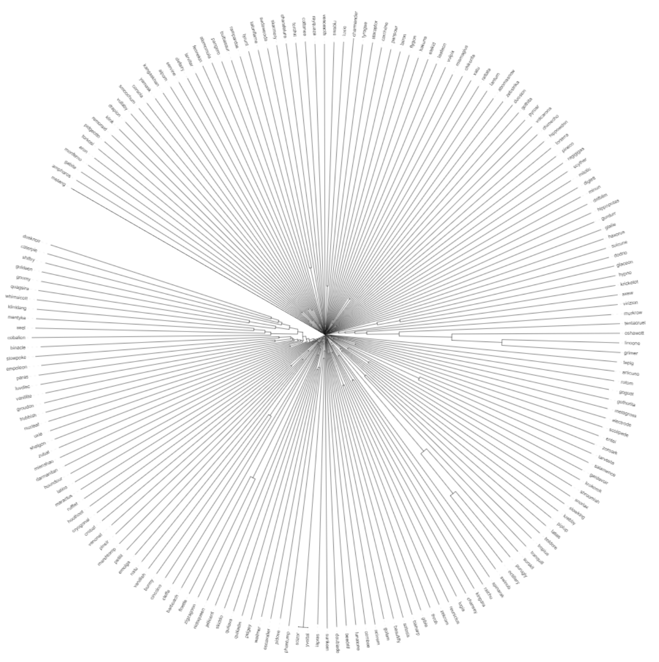
ZPEM1307 Assignment 2 2021



Overview

Biologists frequently look at the myriad species that surround us and attempt to determine their evolutionary history. They are given a fleeting snapshot of biological diversity today, and attempt to infer when different species most recently had common ancestors, hundreds of millions of years into the past.

For this assignment, you will treat a collection of assignments that were submitted by students in a previous year as a set of species, and you will try to infer the evolutionary history of those assignments, to determine which assignments had common ancestors.

You are provided with a set of 187 assignments that were submitted for a single task in a previous year. To protect the identities of the students, the assignments have been renamed and obfuscated using a simple word substitution cypher.

You must choose the level at which you will attempt this assignment. Those who take the A-line will use the neighbor-joining algorithm to infer evolutionary relationships. Those who choose the B-line or C-line will use centroid linkage clustering, which you will recognise from the first assignment. Those who take the B-line and C-line will receive results limited at 84% and 74% respectively. There is a substantial step in difficulty between the three levels, and you may only submit one attempt. Please choose carefully.

Resources

You are provided with

- stubs files of the “aline.py”, “bline.py”, and “cline.py” files;
- unit tests for the functions outlined in those files (“test_aline_public.py”, etc.); and
- the 187 assignments of past students.

The stubs files contain function definitions of the functions that you will need to implement with empty bodies. They also contain scripts that will use your functions to generate dendrograms (and a file, if you take the A-line).

We will refer to your “aline.py”, “bline.py”, or “cline.py” submitted files as “xline.py” (similarly “test_xline.py”, etc) from now on.

Submission

You will submit two files zipped into a single archive, to stop Inspera from mangling their names. They will be named “aline.py” and “test_aline.py”, “bline.py” and “test_bline.py”, or “cline.py” and “test_cline.py”, depending on the level at which you attempt the assignment.

Your “xline.py” file should contain your solutions to the Instructions given below. Your “test_xline.py” file should contain at least one unit test for every function you have modified in “xline.py”.

Rubric

After submission, your marker will

- attempt to run your “xline.py” file in Spyder;
- run your unit tests in “test_xline.py” using “py.test”;
- run the public unit tests (in “test_xline_public.py”); and
- run some private unit tests that will not be available before submission.

If you have implemented the functions as specified, the “main” function in your “xline.py” file should generate a dendrogram when “xline.py” is run in Spyder and it should pass your tests as well as the private and public tests.

Note that the tasks in this assignment could be completed very quickly using packages (SciPy or Scikit-Learn) present in your Anaconda implementation. For that reason you are not permitted to import anything not already imported in the “xline.py” stubs files. Doing so will result in code that uses those imports being removed prior to assessment.

Criterion	Weight	Fail	Pass	Higher Marks
Design	30%	Docstring templates have not been filled in	All docstring templates have been filled and	Code is simple and efficient. B-line

		or are largely incorrect in style or content. Global variables are used. Code is messy and hard to follow.	are concise, clear, and complete. Functions are called appropriately. No global variables are used.	candidates do not unnecessarily calculate dissimilarity. A-line candidates use vectorisation to reduce the number of iterative loops in their code.
Build	40%	Your code fails your unit tests or the public unit tests. Your code fails a significant number of private unit tests. Your code takes more than a minute to complete. Your code does not generate a dendrogram or generates a dendrogram with clear errors.	Your code passes your unit tests and the public unit tests. Your code passes the private unit tests with minor variation. Your code completes in less than 30 seconds and generates a dendrogram.	Your code passes all of the unit tests including the private unit tests. Your code is fast, finishing in well under 30 seconds. Your dendrogram is perfect.
Test	30%	Your tests are not different to the public unit tests or do not test the main specification of the function.	Your tests are different to the public unit tests and test the main specification of every function you have modified.	All of your tests have one-line docstrings explaining their intent. Your tests are neat and well organised.

Instructions

C-Line

You must implement the bodies of the following functions in “cline.py” and fill in their docstrings.

`read(filename, n=3)`

“read” takes a filename as input in a string and reads the appropriate file. The file should first be converted into a list of words, where words are space or line break delimited. The list of words should then be converted into “n-grams” for the input “n”. As an example, if the list of words are “the quick brown fox jumps over the lazy dog” and “n” is 3, the output should be

```
[('the', 'quick', 'brown'),
 ('quick', 'brown', 'fox'),
 ('brown', 'fox', 'jumps'),
 ('fox', 'jumps', 'over'),
 ('jumps', 'over', 'the'),
 ('over', 'the', 'lazy'),
 ('the', 'lazy', 'dog')]
```

Note that the output is a list of tuples.

`vocabulary(ngrams_list)`

“vocabulary” should take a list of lists of tuples as inputs. It should return a list of the unique tuples that were observed. See the public unit tests for an example of how it should function.

`vectorize(ngrams, vocabulary)`

“vectorize” should take a list of n-grams (as obtained from “read”) to vectorize and a list of n-grams (as obtained from “vocabulary”) to use as vocabulary. It should return a 1D NumPy array which is as long as the vocabulary, and contains a 1 if the corresponding tuple in vocabulary is in the “ngrams” input and a 0 otherwise. See the public unit tests for an example of how it should function. (Hint: use a dictionary to speed up this calculation.)

`dissimilarity(v1, v2)`

“dissimilarity” should calculate the cosine dissimilarity between to 1D NumPy arrays, calculated as

$$d(v1,\, v2) \;=\; 1 \;-\; \sum_{i=1}^{n} v1[i] \times v2[i] \,/\!\left(\sum_{i=1}^{n} v1[i]^2 \sum_{i=1}^{n} v2[i]^2\right)^{\frac{1}{2}},$$

where “n” is the length of either vector (which should be the same).

`nearest_pair(vectors, mask)`

“nearest_pair” should take a list of vectors in “vectors” and a list of Boolean values in “mask”, and use “dissimilarity” to find the two nearest vectors such that “mask” is `False` for both. That is, it should return the pair `(i,j)` for which `mask[i]` and `mask[j]` are both `False` and `dissimilarity(vectors[i], vectors[j])` returns the smallest value.

`merge_pair(pair, vectors, linkage, mask)`

“merge_pair” merges the vectors in “vectors” that correspond to the indices in “pair” and appends the merged vector to “vectors”. It also sets “mask” to `True` for those indices and appends `False` to mask. “linkage” is included because it contains the weights to be used for merging the vectors. If “pair” is `(i, j)`, the formula for the merged vector is

```
merged = (linkage[i][3]*vectors[i] + linkage[j][3]*vectors[j]) /
         (linkage[i][3] + linkage[j][3]))
```

`cluster(vectors)`

“cluster” sets up the new lists “mask”, “vectors”, and “linkage”, then iterates over “nearest_pair”, “merge_pair”, and “update_linkage” (“update_linkage” is provided in the stubs file), until all of the vectors are merged. Say that “vectors” contains “n” elements.

“mask” should start as a list of `False` values the same length as “vectors”.

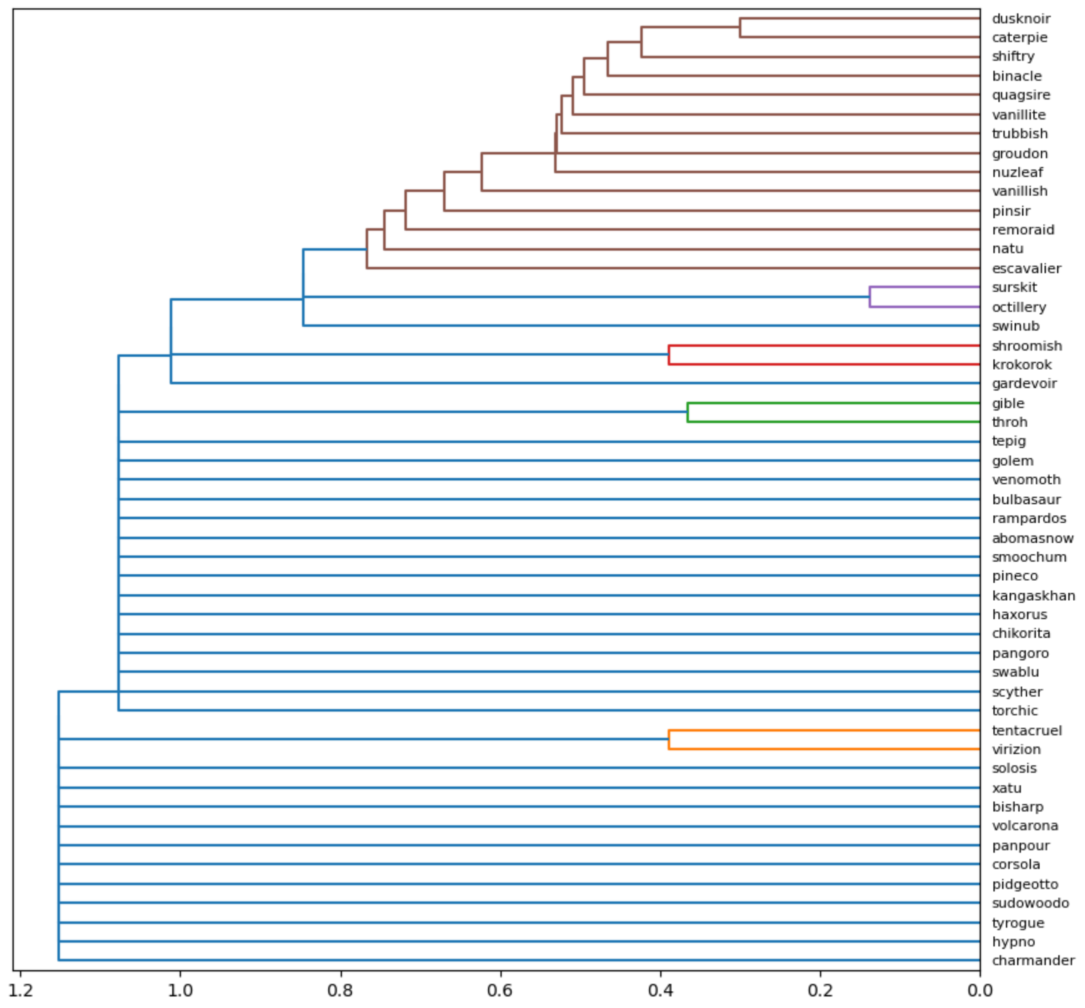
A copy of “vectors” should be created so that the input “vectors” is not changed by the function.

“linkage” should start as `[[None, None, 0, 1]]*n`.

When all the merging is done, “cluster” should return the elements of “linkage” *after* the first “n” (i.e. `linkage[n:]`). That is so that “linkage” complies with the specifications for a `scipy.cluster.hierarchy.linkage` data structure.

Result

When you run “cline.py” in Spyder, you should get a dendrogram that is like the following. Note that the script randomly selects 50 assignments for analysis (because this method is too slow for the whole data set).



B-Line

You must implement the bodies of the following functions in “bline.py” and fill in their docstrings.

`read(filename, n=3)`

“read” takes a filename as input in a string and reads the appropriate file. The file should first be converted into a list of words, where words are space or line break delimited. The list of words should then be converted into “n-grams” for the input “n”. As an example, if the list of words are “the quick brown fox jumps over the lazy dog” and “n” is 3, the output should be

```
[('the', 'quick', 'brown'),
 ('quick', 'brown', 'fox'),
 ('brown', 'fox', 'jumps'),
 ('fox', 'jumps', 'over'),
 ('jumps', 'over', 'the'),
 ('over', 'the', 'lazy'),
 ('the', 'lazy', 'dog')]
```

Note that the output is a list of tuples.

`vocabulary(ngrams_list)`

“vocabulary” should take a list of lists of tuples as inputs. It should return a list of the unique tuples that were observed. See the public unit tests for an example of how it should function.

`vectorize(ngrams, vocabulary)`

“vectorize” should take a list of n-grams (as obtained from “read”) to vectorize and a list of n-grams (as obtained from “vocabulary”) to use as vocabulary. It should return a 1D NumPy array which is as long as the vocabulary, and contains a 1 if the corresponding tuple in vocabulary is in the “ngrams” input and a 0 otherwise. See the public unit tests for an example of how it should function. (Hint: use a dictionary to speed up this calculation.)

`tfidf(vectors)`

“tfidf” should perform a TFIDF transformation on a list of vectors, which should be a list of NumPy arrays. The “term” term in the transformation should just be the number in the input vector (which will be 1 or 0). The IDF term should be the “inverse document frequency smooth”, calculated as the natural log of (the length of “vectors” divided by (the number of times that element was 1 in “vectors” plus 1) plus 1. The formula on Wikipedia is easier to follow. See the public unit tests for an example of its use.

`dissimilarity(v1, v2)`

“dissimilarity” should calculate the cosine dissimilarity between to 1D NumPy arrays, calculated as

$$d(v1,\ v2)\ =\ 1\ -\ \sum_{i=1}^{n} v1[i]\ \times\ v2[i]\ /\left(\sum_{i=1}^{n} v1[i]^2 \sum_{i=1}^{n} v2[i]^2\right)^{\frac{1}{2}},$$

where “n” is the length of either vector (which should be the same).

`distance_matrix(vectors)`

“distance_matrix” should take a list of NumPy arrays and return a 2D NumPy array d where the entry `d[i,j]` should contain the dissimilarity between `vectors[i]` and `vectors[j]`.

`nearest_pair(distances, mask)`

“nearest_pair” takes a distance matrix (as calculated by “distance_matrix”) in “distances” and a list of Boolean values in “mask” as input. “nearest_pair” should return the pair `(i, j)`, where `distances[i,j]` is the smallest value of “distances” such that `mask[i]` and `mask[j]` are both `False`.

`merge_pair(vectors, linkage, mask, pair)`

“merge_pair” should take a pair of indices in “pair”, and merge the corresponding vectors in “vectors” to create a new vector which is appended to “vectors”. It should also append `False` to “mask” and update the “linkage” list.

“linkage” is a list of lists that describes a hierarchical clustering of a set of input vectors. Describe each element of the “linkage” list as a row. If there are “n” data points to be clustered, the first “n” rows of “linkage” correspond to those points in no particular order. Each of those rows looks like this:

```
[None, None, 0, 1]
```

The reason for those entries will become clear in a moment. After the first “n” rows, each row of “linkage” is the result of a merge step, and is defined like this:

```
[child1, child2, distance_to_leaves, number_of_leaves]
```

For each such row, the vectors corresponding to the indices “child1” and “child2” are merged to create the node that corresponds to that row. “distance_to_leaves” is the distance from the merged vector to the observations that were merged into that node. “number_of_leaves” is the number of observations that were merged to create that row.

Under centroid linkage clustering, the distance of a merged tip to the observations that were merged into can sometimes decrease, which can lead to strange-looking trees. So, at each iteration, calculate “distance_to_leaves” as the distance between the merged vectors divided by two plus the average “distance_to_leaves” of the merged vectors. If that is less than either child1’s “distance_to_leaves” or child2’s “distance_to_leaves”, set it to the greater of those.

The new vector that is appended to the vector list can be calculated as the “number_of_leaves” of child1 multiplied by its vector plus the “number_of_leaves” of child2 multiplied by its vector, divided by the sum of those two “number_of_leaves”.

`update_distance_matrix(distances, vectors)`

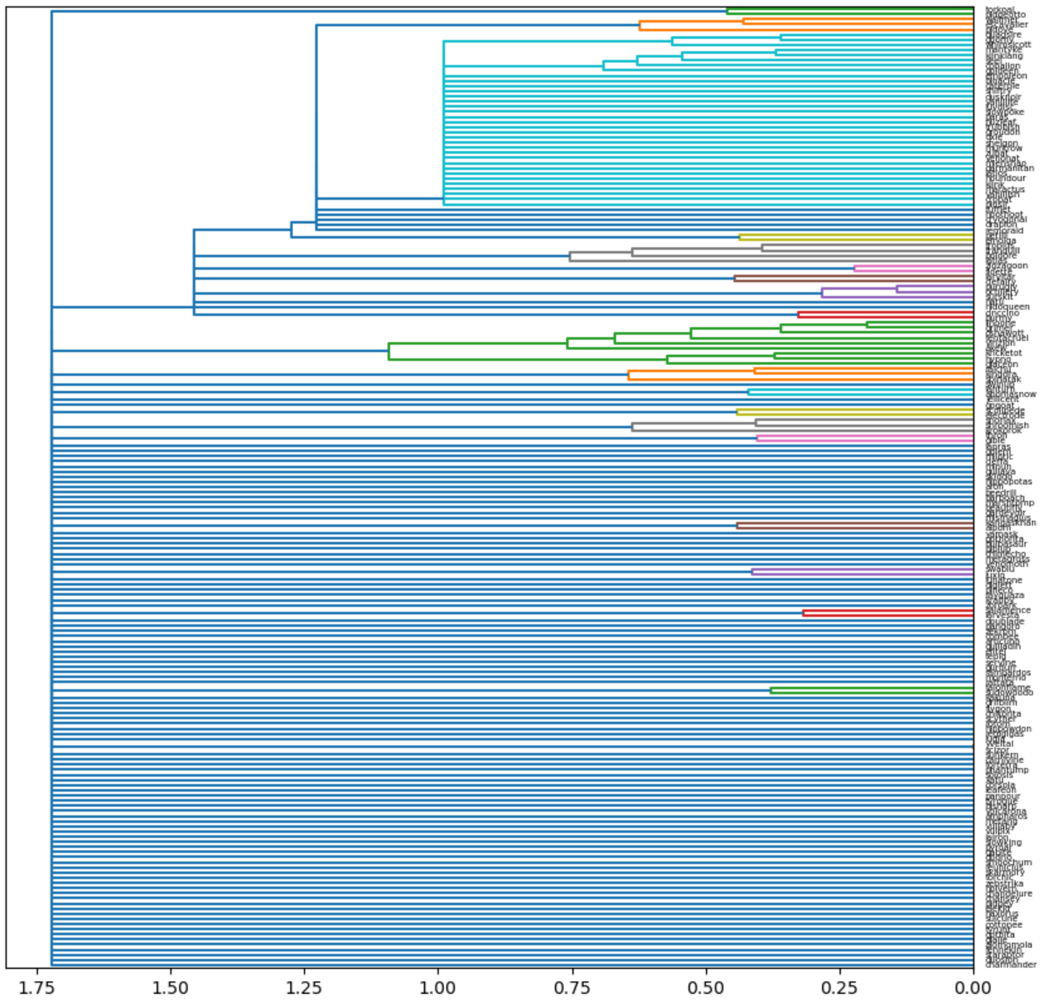
When two clusters are merged, it is necessary to calculate the distance from the merged vector to all of the existing vectors. “update_distance_matrix” takes the existing distance matrix in “distances”, and adds a single column and vector representing the distance of the newly-merged vector to the existing vectors. The newly-merged vector will be the last vector in “vectors”, as created by “merge_pair”.

`cluster(vectors)`

“cluster” should take a list of NumPy arrays, calculate the dissimilarities between them using to form a distance matrix, set up data structures for the “linkage”, and “mask” lists, then iteratively call “nearest_pair”, “merge_pair”, and “update_distance_matrix” until the tree is resolved. Note that the returned value of “linkage” should then correspond to the description of a linkage array as returned by `scipy.cluster.hierarchy.linkage`. That is, it should only include the rows that correspond to merged vectors.

Result

When you run “bline.py” in Spyder, you should get a dendrogram that is like the following.



A-Line

You must implement the bodies of the following functions in “aline.py” and fill in their docstrings.

`read(filename, n=3)`

“read” takes a filename as input in a string and reads the appropriate file. The file should first be converted into a list of words, where words are space or line break delimited. The list of words should then be converted into “n-grams” for the input “n”. As an example, if the list of words are “the quick brown fox jumps over the lazy dog” and “n” is 3, the output should be

```
[('the', 'quick', 'brown'),
 ('quick', 'brown', 'fox'),
 ('brown', 'fox', 'jumps'),
 ('fox', 'jumps', 'over'),
 ('jumps', 'over', 'the'),
 ('over', 'the', 'lazy'),
 ('the', 'lazy', 'dog')]
```

Note that the output is a list of tuples.

`vocabulary(ngrams_list)`

“vocabulary” should take a list of lists of tuples as inputs. It should return a list of the unique tuples that were observed. See the public unit tests for an example of how it should function.

`vectorize(ngrams, vocabulary)`

“vectorize” should take a list of n-grams (as obtained from “read”) to vectorize and a list of n-grams (as obtained from “vocabulary”) to use as vocabulary. It should return a 1D NumPy array which is as long as the vocabulary, and contains a 1 if the corresponding tuple in vocabulary is in the first input and a 0 otherwise. See the public unit tests for an example of how it should function. (Hint: use a dictionary to speed up this calculation.)

`tfidf(vectors)`

“tfidf” should perform a TFIDF transformation on a list of vectors, which should be a list of NumPy arrays. The “term” term in the transformation should just be the number in the input vector (which will be 1 or 0). The IDF term should be the “inverse document frequency smooth”, calculated as the natural log of (the length of “vectors” divided by (the number of times that element was 1 in “vectors” plus 1) plus 1. The formula on Wikipedia is easier to follow. See the public unit tests for an example of its use.

`dissimilarity(v1, v2)`

“dissimilarity” should calculate the cosine dissimilarity between to 1D NumPy arrays, calculated as

$$d(v1,\ v2)\ = 1\ -\ \sum_{i=1}^{n} v1[i] \times v2[i] \,/ \left(\sum_{i=1}^{n} v1[i]^2 \sum_{i=1}^{n} v2[i]^2 \right)^{\frac{1}{2}},$$

where “n” is the length of either vector (which should be the same).

`distance_matrix(vectors)`

“distance_matrix” should take a list of NumPy arrays and return a 2D NumPy array d where the entry `d[i,j]` should contain the dissimilarity between `vectors[i]` and `vectors[j]`.

`nearest_pair(distances)`

Assuming that “distances” is a distance matrix as calculated by “distance_matrix”, nearest_pair should find the two vectors that correspond to the nearest nodes as calculated according to the neighbor-joining algorithm of Saitou and Nei (1987). Note that the nearest nodes may not be the ones with the smallest distance between them, and that a transformation is required.

`update_distance_matrix(distances, pair)`

“update_distance_matrix” should take a distance matrix as calculated by “distance_matrix” or a previous call to “update_distance_matrix”, and should calculate a new distance matrix where the pair of indices in “pair” are merged into a new node according the neighbor-joining algorithm. That is, the columns and rows corresponding to the indices in “pair” should be removed, and a new row and column should be added to the resulting matrix, which describes the distances from the remaining nodes to the new node.

“update_distance_matrix” should then return the distances from the new node to the pairs of nodes that were merged in a tuple and the new distance matrix. See the public unit tests for an example of its use.

`merge_pair(graph, linkage, pair_distances, ix, pair)`

“merge_pair” should merge the two nodes in “pair” and update “graph”, “linkage”, and “ix”. “pair_distance” should contain the distances from “update_distance_matrix”.

“linkage” is a list of lists that describes a hierarchical clustering of a set of input vectors. Describe each element of the “linkage” list as a row. If there are “n” data points to be clustered, the first “n” rows of “linkage” correspond to those points in no particular order. Each of those rows looks like this:

```
[None, None, 0, 1]
```

The reason for those entries will become clear in a moment. After the first “n” rows, each row of “linkage” is the result of a merge step, and is defined like this:

```
[child1, child2, distance_to_leaves, number_of_leaves]
```

For each such row, “child1” and “child2” are merged to create the node that corresponds to that row. “distance_to_leaves” is the distance from that node to the observations that were merged into that node. “number_of_leaves” is the number of observations that were merged to create that row.

Sadly, the neighbor-joining algorithm will give you different distances from a given merged node to the leaves that were merged into it, depending on the leaf. So, at each iteration, calculate “distance_to_leaves” as the average of (the distance from the merged node to child1 plus child1’s “distance_to_leaves”) and (the distance from the merged node to child2 plus child2’s “distance_to_leaves”). If that is less than either child1’s “distance_to_leaves” or child2’s “distance_to_leaves”, set it to the greater of those.

“number_of_leaves” is just the sum of the “number_of_leaves” of child1 and child2.

As “linkage” does not perfectly hold the outcome of the neighbor-joining algorithm, at each merge step add an edge to the `networkx.Graph` contained in “graph”. Set its “length” attribute to be the distance to the appropriate child.

Finally, “ix” should be used to keep track of which observations and nodes have already been merged. It should contain the indices of merged nodes or observations that have **not** yet been merged into other nodes. It should be updated by “merge_pair”. Note that “ix” is also necessary to map between the every-changing distance matrix and positions of your “linkage” list.

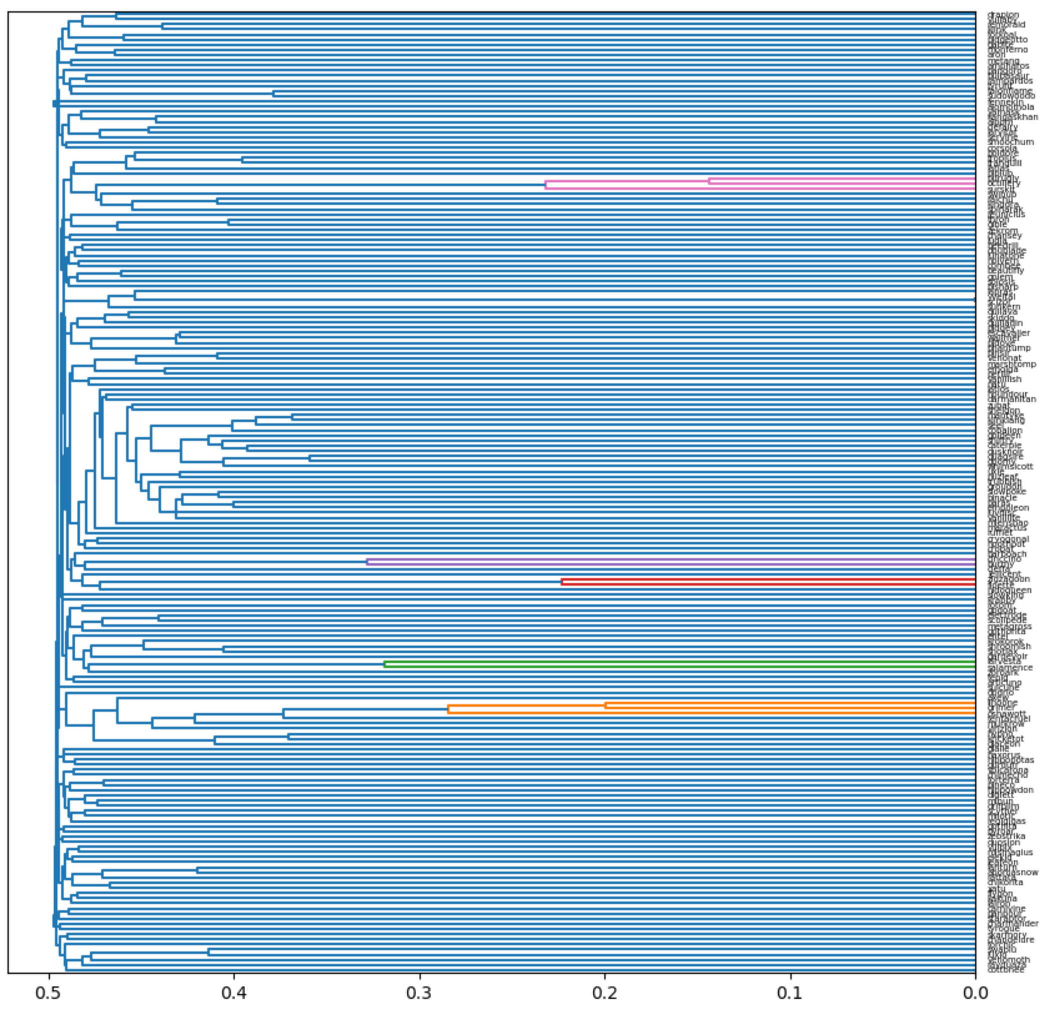
See the public unit tests for a usage example, and the linkage format will be presented in lectures over the coming weeks.

`cluster(vectors)`

“cluster” should take a list of NumPy arrays, calculate the dissimilarities between them using to form a distance matrix, set up data structures for the “graph”, “linkage”, and “ix” data structures, then iteratively call “nearest_pair”, “update_distance_matrix”, and “merge_pair” until the tree is resolved. It should then return “graph” and all the merged elements of “linkage”. Note that the returned value of “linkage” should then correspond to the description of a linkage array as returned by [`scipy.cluster.hierarchy.linkage`].

Result

When you run “aline.py” in Spyder, you should get a dendrogram that is like the following. The script will also save a file called “tree.tree” in your working directory, which you can feed to a website like https://itol.embl.de/ to get a prettier dendrogram (like the one at the start of this document).


0.5
0.4
0.3
0.2
0.1
0.0